Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 557 – 566

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

A Fast Algorithm to Find All-Pairs Shortest Paths in Complex Networks

Wei Peng[1], Xiaofeng Hu, Feng Zhao, Jinshu Su

School of Computer,National University of Defense Technology
Changsha, Hunan, 410073, China

Abstract

Finding shortest paths is a fundamental problem in graph theory, which has a large amount of applications in many areas like computer science, operations research, network routing and network analysis. Although many exact and approximate algorithms have been proposed, it is still a time-consuming task to calculate shortest paths for large-scale networks with tremendous volume of data available in recent years. In this paper, we find that the classic Dijkstra's algorithm can be improved by simple modification. We propose a fast algorithm which utilize the previously-calculated results to accelerate the latter calculation. Simple optimization strategies are also proposed with consideration of characteristics of scale-free complex networks. Our experimental results show that the average running time of our algorithm is lower than the Dijkstra's algorithm by a factor relating to the connection probability in random networks of ER model. The performance of our algorithm is significantly better than the Dijkstra's algorithm in scale-free networks generated by the AB model. The results show that the time complexity is reduced to about $O(n^{2.4})$ in scale-free complex networks. When the optimization strategies are applied, the algorithm performance is further improved slightly in scale-free networks.

Keywords: All-Pairs Shortest-Paths Problem, Complex Network, Algorithm, Time Complexity

1. Introduction

Finding shortest paths is a classic problem in graph theory. It has enormous applications in many areas including computer science, operations research, transportation engineering, network routing and network analysis. Due to its great values in both scientific research and engineering tasks, it has been extensively studied in all kinds of classic graphs or networks, directed or undirected, weighted or unweighted.

There are algorithms with polynomial time complexities for the shortest path problems. For directed graphs with real edge weights, the best-known algorithm [1] for the all-pairs shortest-path (APSP) problem has the time complexity of $O(n^3/\log n)$. For unweighted undirected graphs, the APSP problem can be solved in $O(nm)$ time [2]. Although these algorithms are efficient, they are still time-consuming when applied on large-scale complex networks.

Email addresses: wpeng@nudt.edu.cn (Wei Peng), xfhu@nudt.edu.cn (Xiaofeng Hu), fengzhao@nudt.edu.cn (Feng Zhao), sjs@nudt.edu.cn (Jinshu Su)

[1]Corresponding author

doi:10.1016/j.procs.2012.04.060

Complex networks have attracted much attention in recent years. It has been found that many systems in real world can be modeled as a large-scale complex network and they can be studied using random graph theory. Such networks include the Internet, electronic grid, transport network, metabolic network and social network. Due to their large-scale feature, it becomes a critical task to study their characteristics by calculating the shortest paths in them. Many approximating shortest-path algorithms have been proposed to meet the requirements of large-scale complex network analysis [3][4]. However, it is still necessary to find the exact shortest-paths in many applications.

In this paper, we find that the classic Dijkstra's algorithm [5] can be improved with simple modification. We propose an algorithm for the APSP problem and optimization strategies with consideration of the characteristics of scale-free complex networks. Then the performance of the proposed algorithms is studied by experiments on random networks generated by the Erdös-Rényi (ER) model [6] and the Albert-Barabási (AB) model [7]. With different parameter settings, the experiment results show that the proposed algorithm has better performance than the Dijkstra's algorithm. The results are significantly better when applied on scale-free complex networks. Using regression methods, we find that the time complexity of our algorithm is only about $O(n^{2.4})$ while the original Dijkstra's algorithm has a time complexity of $O(n^3)$ for the APSP problem. The experimental results also show that the performance of the proposed algorithm is slightly improved in scale-free complex networks when the optimization strategies are applied.

The rest of the paper is organized as follows. Section II presents a short survey on the related research work. Section III describes the shortest-path algorithms and optimization strategies for scale-free complex networks. The experimental results are presented in section IV. Finally, we conclude the paper in section V.

2. Related Work

The shortest-paths problems are classic problems in graph theory and there are many research results toward designing an efficient algorithm to calculate shortest paths in all kinds of networks. Two classic algorithms for the single-source shortest-path (SSSP) problem are the Bellman-Ford algorithm [8] and the Dijkstra's algorithm [5]. They have $O(nm)$ and $O(n^2)$ time complexities, respectively, where n is the number of vertices and m is the number of edges in a graph. The Dijkstra's algorithm is used in a network where each edge has a positive length (or weight) value. The Bellman-Ford algorithm can be applied in the situation when edge lengths are negative. For the APSP problem, the Floyd-Warshall algorithm is a classic algorithm which has the time complexity of $O(n^3)$.

Many algorithms have been proposed by improving or combining the above classic algorithms. For example, based on the Dijkstra's algorithm, Orlin et al. [9] proposed a faster algorithm for the SSSP problem in networks with few distinct positive lengths. Goldberg et al. [10] proposed an efficient shortest path algorithm which combines with A^* search. Roditty and Zwick [11] studied the dynamic shortest path problems and proposed a randomized fully-dynamic algorithm for the APSP problem in directed unweighted graphs. For directed graph with real edge weights, Pettie's algorithm [12] solves the APSP problem in $O(nm + n^2\text{loglog}n)$ time. Chan [1] firstly obtained an algorithm with time complexity $O(n^3/\log n)$ in 2005, which is the best-known result for the APSP problem in a directed graph. Han [13] also proposed an algorithm with the same time complexity later, using a smaller table.

To meet the requirements of large-scale complex network analysis, many algorithms have recently been proposed to find the approximating solutions for the shortest-paths problems. Baswana et al. [3] propose a randomized algorithm for computing all-pairs nearly 2-approximate distances with time complexity of $O(n^2\log^{3/2}n)$. The approximate distance is bounded by $2\delta(u,v) + 3$ where $\delta(u,v)$ is the shortest-path distance between u and v. Potamias et al. [14] propose a landmark-based method to estimate point-to-point distances in very large networks. Similarly, Tang et al. [4] propose an approximate algorithm for the APSP problem in complex networks. The algorithm first selects some high-degree nodes as local central points and find the approximate shortest-path distances using the pre-calculated shortest-paths starting from these local central points. In contrast to these works, we try to find simple and efficient methods to calculate the exact shortest-path distances in complex networks.

3. New Algorithms for All-Pairs Shortest-Paths Problem

3.1. Basic Algorithm

The well-known Dijkstra's algorithm uses a simple breadth-first search approach to find all shortest paths from a single source to all other vertices in a given graph. It has the advantages of efficiency and simplicity. To find all-pairs

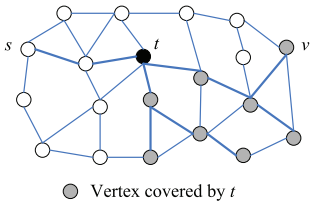


Figure 1: Example of covered nodes

shortest-paths, we can apply the Dijkstra's algorithm on each vertex iteratively. Such an intuitive approach is simple, but not very efficient.

To improve the Dijkstra's algorithm on the APSP problem, we propose a new algorithm that utilizes information obtained in previous steps to accelerate the latter process. Let $G = (V, E)$ denote a weighted directed graph, where V and E are the set of vertices and edges respectively. The edge from the vertex u and v is denoted as (u, v) and its weight is $w(u, v)$. We give the following definition to illustrate the main idea of our algorithm.

Definition 1: If the vertex t is an intermediate vertex on the shortest path from the vertex u to v, then we say that v is *covered* by t.

The Dijkstra's algorithm uses a breadth-first search (BFS) method to get shortest paths starting from a single source. When the vertex t is visited in the search process, if the shortest paths from t have been obtained in previous steps, then we can immediately obtain the shortest paths from source s to all other vertices using t as an intermediate vertex. In other words, given a source s, we need not visit other vertices which are covered by t. Based on this idea, we propose the following algorithm which improves the Dijkstra's algorithm on solving the APSP problem. Many vertices may be covered by an intermediate vertex t (see Figure 1 for an example), so the total time used may be reduced dramatically.

We use the following data structures in the proposed algorithm:

- L: the matrix containing the edge weights, where $L[u, v]$ is the weight of edge (u, v). If the edge (u, v) does not exist, then $L[u, v] = \infty$;
- D: the distance matrix, where $D[u, v]$ is the distance from the vertex u to v. Initially, $D[u, v] = \infty$ for all vertex pairs;
- $flag$: the vector to indicate whether the shortest paths from a vertex to other vertices have been calculated. All elements of the vector are set to zero initially. After the shortest paths for vertex u are calculated, $flag[u]$ is set to 1.
- Q: the min-priority queue containing the vertices to be visited. It is the same queue as that used in the classic Dijkstra's algorithm.

To find all shortest paths in a graph G, the algorithm firstly initializes the flag vector by setting all elements as zero (step 1-2 in Algorithm 1) and initializes all elements of the distance matrix to be infinity (step 3-4 in Algorithm 1). Then the modified Dijkstra's procedure (Procedure 1) is called iteratively to find shortest paths starting from every vertex.

Compared to the classic Dijkstra's algorithm, the steps 5-10 in procedure 1 are main stuff newly added. When a vertex t is visited, if the shortest paths starting from it have been obtained ($flag[t]$ is set to 1) (step 5 in Procedure 1), then the shortest paths from the source s to other vertices are updated by using t as an intermediate vertex (step 6-10 in Procedure 1). After all shortest paths starting from s have been obtained, we set $flag[s]$ to 1 to indicate it. Other steps are the same as those in the classic Dijkstra's algorithm. The procedure $Enqueue(Q, v)$ adds a vertex v in the min-priority queue Q. The procedure $DeQueue(Q)$ gets a vertex from the queue Q which has the smallest shortest-path starting from s.

Procedure 1: Modified Dijkstra's Procedure
Input: Graph $G = (V, E)$, source s, weight matrix L, distance matrix D, vector $flag$
Output: updated distance matrix D, updated vector $flag$

```
1:  D[s, s] = 0
2:  Q = {s}
3:  while Q is not empty do
4:      t=DeQueue(Q)
5:      if flag[t] = 1, then
6:          for each vertex v ∈ V do
7:              if D[s, t] + D[t, v] < D[s, v] then
8:                  D[s, v]=D[s, t] + D[t, v]
9:              end if
10:         end for
11:     else
12:         for each edge (t, v) outgoing from t do
13:             if D[s, t] + L[t, v] < D[s, v] then
14:                 D[s, v]=D[s, t] + L[t, v]
15:                 Enqueue(Q,v)
16:             end if
17:         end for
18:     end if
19: end while
20: flag[s] = 1
```

Algorithm 1: Basic Algorithm for the APSP Problem
Input: Graph $G = (V, E)$, weight matrix L
Output: distance matrix D

```
1: for each vertex v ∈ V do
2:     flag[v] = 0
3: for each vertex pair (u,v) do
4:     D[u, v] = ∞
5: for each vertex v ∈ V do
6:     call the modified Dijkstra's procedure
       to get shortest paths starting from v
```

The following lema ensures the correctness of the algorithm.

Lemma 1: If the vertex t is an intermediate vertex on the shortest path from the source s to u, and u is an intermediate vertex on the shortest path from s to v, then the vertex t is also an intermediate vertex on the shortest path from s to v.

Proof: Assume the shortest path from s to u is $(s, \ldots, t, \ldots, u)$ and the shortest path from s to v is $(s, \ldots, u, \ldots, v)$. Because sub-paths of a shortest path are also shortest paths, the shortest path from s to v can be obtained by concatenating $(s, \ldots, t, \ldots, u)$ and the sub-path $(u, \ldots, v)$ in the shortest path from s to v. Thus, the vertex t will be in the shortest path from s to v. ■

Theorem 1: Given a graph $G = (V, E)$ and weights of edges, the algorithm 1 finds the shortest paths for all vertex pairs.

Proof: We prove that the modified Dijkstra's procedure finds shortest paths starting from a source s.

Assume the predecessor of a vertex v on the shortest path from s to v is u. If u is not covered by any vertex which shortest paths have been found, then the shortest path from s to v is obtained by the Dijkstra's BFS procedure (steps 12-17). If u is covered by an intermediate vertex t which shortest paths have been obtained, then t will be an intermediate vertex on the shortest path from s to v, too, according to Lemma 1. Therefore, the shortest path from s to v will be obtained when the vertex t is visited (steps 6-10). Thereby, the modified Dijkstra's procedure will find the shortest path from s to v, and the algorithm 1 will find shortest paths for all vertex pairs. ■

The newly-added steps do not change the upper bound on the time complexity of the algorithm. The proposed algorithm has the same time complexity than the Dijkstra's algorithm. Assume the time complexity of operations on the min-priority queue is δ, then the time complexity of proposed algorithm is $O(n(\delta n + m))$. For directed graph with real edge weights, if the queue is implemented simply as an array, then $\delta = O(n)$ and the time complexity of the algorithm is $O(n^3 + nm) = O(n^3)$. If the queue is implemented with a Fibonacci heap, then $\delta = O(\log n)$ and the time complexity of the algorithm is $O(n^2 \log n + nm)$. For unweighted undirected graph, $\delta = O(1)$ and the time complexity of the algorithm is $O(n^2 + nm) = O(nm)$.

3.2. Optimization for Complex Networks

Complex networks exhibit some interesting features which can be utilized to optimize the procedure of finding shortest paths. Two most important features are called "small-world" [15] and "scale-free" [16], respectively. The small-world feature means that nodes in a complex network can be highly clustered, while having small shortest path lengths from each other. The scale-free feature indicates that degrees of vertices in a complex network follow a scale-free power-law distribution. Intuitively, the network is non-uniform and its connectivity is dominated by relatively few high-degree nodes.

Since few nodes in a complex network have large number of neighbors (neighboring nodes), it is very possible for these nodes to be in the middle of shortest paths of other nodes. If the shortest paths from these high-degree nodes are obtained in advance, then the visiting of other nodes covered by them can be saved in the modified Dijkstra's procedure. Therefore, the order of vertices to be selected as sources is important for the algorithm performance in the context of complex networks.

To determine the order of vertices as sources, one can simply sort the vertices by their degrees. For example, we can sort vertices in descending order and select vertices with high degrees as sources before vertices with low degrees.

Because a large number of vertices in a complex network have low degrees, we need not determine the order for all vertices so that the time complexity of the selection process can be reduced. For example, we can use a ratio parameter r ($0 < r < 1$) to control the selection process. If nr nodes have been selected, then the selection process can be stopped and the vertices which have not been selected will be used as sources in a random order.

Based on above ideas, we propose an optimized algorithm for the APSP problem. The following data structures are added:

- *deg*: the vector containing the degrees of vertices. $deg[i]$ is the degree of the i-th vertex;
- *order*: the vector containing the indices of vertices to be used as sources. $order[i]$ is the index of i-th source vertex.

The optimized algorithm is illustrated by Algorithm 2.

The steps 5-6 are used to calculate degrees of all vertices and vertices are selected according to their degrees in steps 7-12. The $swap(a, b)$ operation swaps the values of two variables a and b.

Calculating degrees of all vertices requires $O(m)$ time. The selection procedure has the time complexity of $O(rn^2)$. Thus, the newly-added steps have the time complexity of $O(m + rn^2)$.

Algorithm 2: Optimized Algorithm for the APSP Problem
Input: Graph $G = (V, E)$, weight matrix L, ratio r
Output: distance matrix D

```
1:  for each vertex v ∈ V do
2:      flag[v] = 0
3:  for each vertex pair (u,v) do
4:      D[u, v] = ∞
5:  for i = 1 to n do
6:      Calculate the degree of the i-th vertex
7:  for i = 1 to n do
8:      order[i] = i
9:  for i = 1 to rn do
10:     for j = i + 1 to n do
11:         if deg[order[j]] > deg[order[i]] then
12:             swap(order[j], order[i])
13: for i = 1 to n do
14:     call the modified Dijkstra's procedure
        using the source of index order[i]
```

Algorithm 3: Adaptive Algorithm for the APSP Problem
Input: Graph $G = (V, E)$, weight matrix L
Output: distance matrix D

```
1:  for each vertex v ∈ V do
2:      flag[v] = 0
3:  for each vertex pair (u,v) do
4:      D[u, v] = ∞
5:  for i = 1 to n do
6:      Calculate the degree of the i-th vertex
7:  for i = 1 to n do
8:      order[i] = i
9:  for i = 1 to n do
10:     for j = i + 1 to n do
11:         if deg[order[j]] > deg[order[i]] then
12:             swap(order[j], order[i])
13:         end if
14:     end for
15:     call the modified Dijkstra's procedure
        using the source of index order[i]
```

3.3. Adaptive Optimization Algorithm

In algorithm 2, we use node degrees to determine the order of vertices to be used as sources. Although it is possible for nodes with high degrees to be in the middle of shortest paths of others, we can not guarantee that the optimization in algorithm 2 is always effective. Indeed, a better mechanism is required to improve the algorithm further.

An interesting fact is that there are heuristic information in the modified Dijkstra's procedure and those information can be utilized to help selecting sources. In steps 12-17 of the modified Dijkstra's procedure, if the condition in step 13 is true, then the shortest path from souce s to a vertex v may traverse the edge (t, v) temporarily. It hints at that the vertex t is a possible intermediate vertex on some shortest paths. If the vertex t is used as predecessor more frequently, it will be possible for it to cover more vertices. Thus, it should be used as source in advance.

Based on the observations, we propose an improved algorithm which can select sources adaptively. For a vertex t, we use the variable $deg[t]$ to store its priority to be selected as the source. The variable $deg[t]$ is initially set to the degree of t, and later updated in the modified Dijkstra's procedure. If the condition in step 13 of the modified Dijkstra's procedure is true, then we let

$$deg[t] = deg[t] + c \tag{1}$$

where c is a predefined constant, e.g., $c = 1$. Other steps in the modified Dijkstra's procedure are not changed.

The main procedure is stated in Algorithm 3. Since the vector deg changes in the modified Dijkstra's procedure, the algorithm will select vertices as sources adaptively.

Calculating degrees of all vertices requires $O(m)$ time. The selection procedure in steps 10-14 has the time complexity of $O(n)$. Thus, the algorithm 3 has the same time complexity than the algorithm 1.

4. Evaluation of the Algorithms

4.1. Introduction of Experiments

We evaluate the performance of the proposed algorithms on random networks. We compare their performance in complex networks with the Erdös and Rényi (ER) random graph model [6] and the scale-free Albert-Barabási (AB) network model [7].

In the ER model, each vertex pair is uniformly connected with a probability p in a network of n nodes. There are about $pn(n-1)/2$ edges in a ER graph. The average degree of vertices is $\langle k \rangle = p(n-1) \approx pn$. The degrees of vertices can be represented by the Poission distribution.

The AB model is an extension of the original Barabási-Albert (BA) model. In the model, a network initially contains m_0 nodes. Then the network grows using the following operations [7]:

1. With probability p, add m new edges. For each edge, one of its end-points is selected randomly from the existing nodes. Another end-point is selected with probability

$$\Pi(k_i) = \frac{k_i + 1}{\sum_j k_j + 1} \tag{2}$$

   where k_i is the degree of the i-th node;
2. With probability q, rewire m edges selected randomly in the network. For each edge, one of its end-points is changed to another node which is selected with a probability given by the Equation (2);
3. With probability $1 - p - q$, add a new node and m new edges connecting the new node to other existing nodes. Another end-point of each new edge is also selected with a probability given by the Equation (2).

Depending on the model parameter values, the AB model can not only generate networks with power-law degree distributions, but also networks with exponential degree distributions. When $q < 0.5$, networks with power-law degree distributions are generated. The model parameter values used in our experiments are given in Table 1.

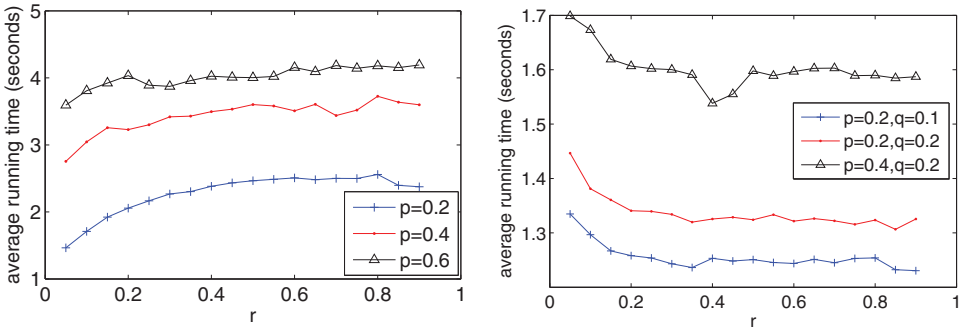
The algorithms to be compared include the classic Dijkstra's algorithm which is called iteratively using every vertex as source, algorithm 1, algorithm 2 and algorithm 3.

For each parameter setting, we randomly generate 50 network instances and run the algorithms on them. Then, the running time is averaged on all instances.

The test platform is a laptop with Intel Core i7-2640M CPU and 4GB memory, running Fedora 16 (Linux core 3.1.0-7) operating system.

Table 1: Parameter Settings in Experiments

Parameter	*Value*
number of vertices in a network (n)	500-5000
Probability p in the ER model	0.2,0.5,0.8
Parameter m_0 in the AB model	10
Parameter m in the AB model	2
Probability p in the AB model	0.2,0.3,0.8
Probability q in the AB model	0.1,0.2
ratio r in the algorithm 2	0.25
constant c in the algorithm 3	1



(a) ER model (b) AB model

Figure 2: Effects of parameter r

4.2. Effects of Algorithm Parameters

We test the performance of algorithms on different algorithm parameter values. The node number is set to 1000 for the ER model and 3000 for the AB model.

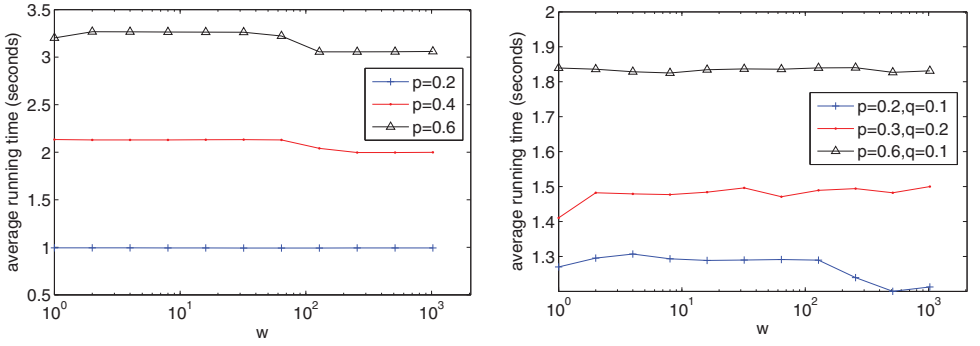
With different values of parameter r, the performance of algorithm 2 is demonstrated in Figure 2. It is shown that the average running time slightly increases with the increasing of r in cases of ER model. It seems that the optimization strategy is not useful in random networks of the ER model. In scale-free networks of the AB model, however, the average running time decreases with increasing of r. The average running time does not change significantly when $r > 0.3$. Thus, we can use a small value of r in algorithm 2 to improve the algorithm performance in scale-free networks.

The performance of algorithm 3 with different values of parameter c is demonstrated in Figure 3. The average running time does not change much in most cases. Therefore, we can assume that the value of c has little impact on the performance of the algorithm 3.

4.3. Comparison of Algorithm Performance

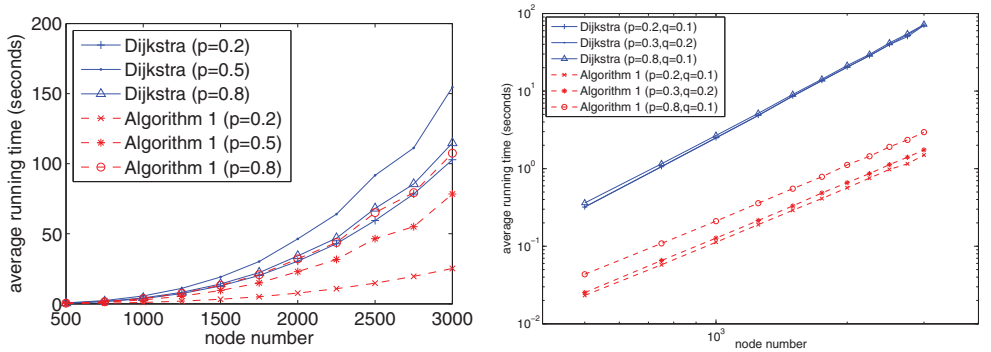
We compare the performance of the classic Dijkstra's algorithm and the algorithm 1 on random networks of ER model and AB model.

From Figure 4a, we can see that the average running time increases with the increasing of node number. The algorithm 1 outperforms than the Dijkstra's algorithm in random networks of ER model with different link probability. Both algorithms have the same level of time complexity which is $O(n^3)$, but with different factors. Let the time complexity of Dijkstra's algorithm be $O(a_0 n^3)$ and the time complexity of algorithm 1 be $O(a_1 n^3)$. Using polynomial regression method, we obtain the values of a_0 and a_1 in each case. The results are shown in Table 2. From the table, we can see that the coefficient ratio (a_1/a_0) increases with the increasing of probability p. When $p = 0.8$, the average running time of the algorithm 1 is about 85% of the time of the Dijkstra's algorithm. When p drops down to 0.2, the



(a) ER model (b) AB model

Figure 3: Effects of parameter c



(a) ER model (b) AB model

Figure 4: Comparison with Dijkstra's Algorithm

algorithm 1 takes only about 22% running time of the Dijkstra's algorithm. Thereby, our algorithm is very efficient in cases of sparse random networks.

Figure 4b shows the *log-log* curve of the average running time in scale-free networks of AB model. Interestingly, we find that the performance of algorithm 1 is significantly better than the performance of the Dijkstra's algorithm. The time complexity of algorithm 1 is even reduced in scale-free networks. From figure 4b, we can see that the average running time follows the paw-law distribution for both algorithms. Denote the average running time by y and the node number by n. The relationship between them can be expressed by:

$$log(y) = b_0 + b_1 log(n) \tag{3}$$

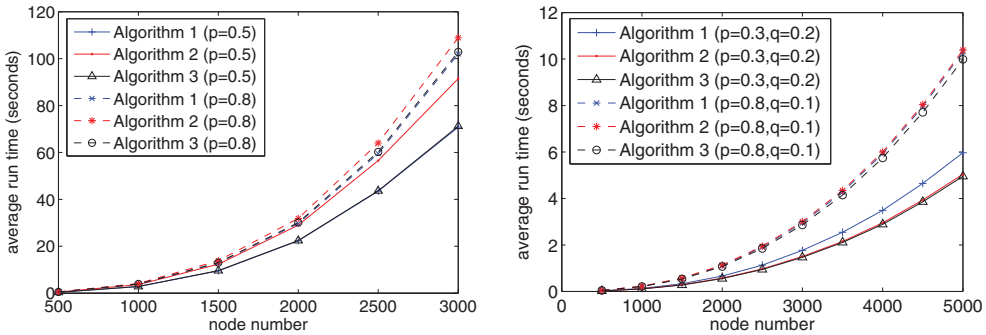
where b_0 and b_1 are coefficients. The values of b_0 and b_1 are calculated using the linear regression method and the results are shown in Table 3. It is shown that the time complexity of Dijkstra's algorithm is still $O(n^3)$, but the time

Table 2: Coefficients on Cases of ER Model

Case	a_0	a_1	a_1/a_0
$p = 0.2$	3.94856×10^{-9}	0.88043×10^{-9}	0.2230
$p = 0.5$	5.82049×10^{-9}	3.28129×10^{-9}	0.5637
$p = 0.8$	4.36906×10^{-9}	3.73352×10^{-9}	0.8545

Table 3: Coefficients on Cases of AB Model

	Dijkstra's Algorithm		Algorithm 1	
Case	b_0	b_1	b_0	b_1
$p = 0.2, q = 0.1$	-19.80237	3.00059	-18.22028	2.32293
$p = 0.3, q = 0.2$	-19.84295	3.00704	-18.37842	2.36342
$p = 0.8, q = 0.1$	-19.45874	2.96218	-17.83301	2.35931



(a) ER model (b) AB model

Figure 5: Comparison of Proposed Algorithms

complexity of Algorithm 1 is reduced to about $O(n^{2.4})$.

The performance of three proposed algorithms is compared at last. The results are shown in Figure 5. It can be observed that the average running time of algorithm 2 is slightly higher than other two algorithms in cases of ER model, while the performance of algorithm 3 is comparable to that of algorithm 1. However, in cases of AB model, the average running time of algorithm 2 and the time of algorithm 3 are lower than the time of algorithm 1 when p is low. When p is high, the performance of algorithm 2 is comparable to the performance of algorithm 1 while the performance of algorithm 3 is slightly better than the other two algorithms.

5. Conclusion

In this paper, we have studied the all-pairs shortest-paths (APSP) problem. We propose a simple and efficient algorithm by modifying the classic Dijkstra's algorithm for the APSP problem. The idea of the algorithm is simple. That is, we use the shortest-path results calculated in previous steps to help finding the shortest paths in latter steps. Considering the scale-free feature of complex networks, we also proposed optimization strategies to improve the algorithm performance. Experiments are conducted to evaluate the proposed algorithms. The results show that our algorithm is significantly better than the Dijkstra's algorithm in sparse networks and scale-free complex networks. The time complexity is only about $O(n^{2.4})$ when our algorithm is applied in scale-free networks generated by the AB model. The algorithm performance is slightly improved with the optimization strategies in scale-free networks.

Our algorithm is useful in finding exact shortest-path distances in complex networks. It also provides a good base to design more efficient algorithms in the research of complex networks. For example, the algorithm may be modified to calculate the approximate shortest paths and other metrics in complex networks.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China under grant No.61070199, 61103189 and 61100223, and Hunan Provincial Natural Science Foundation of China under grant No.11JJ7003.

References

[1] T. M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. in: Proc. 9th Workshop Algorithms Data Structures, in: Lecture Notes in Computer Science, vol.3608, Springer, Berlin, 2005, pp.318-324.

[2] T. M. Chan. All-Pairs Shortest Paths for Unweighted Undirected Graphs in $o(mn)$ Time. SODA'06, January 22-26, Miami, FL, 2006.

[3] S. Baswana, V. Goyal, S. Sen. All-Pairs nearly 2-approximate shortest paths in $O(n^2 polylogn)$ time. Theoretical Computer Science, vol.410, no.1, pp.84-3, 2009.

[4] J.-T. Tang, T. Wang, and J. Wang. Shortest Path Approximate Algorithm for Complex Network Analysis. Journal of Software, vol.22, no.10, pp.2279-2290, 2011.

[5] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. Numerische Mathematik, vol.1, pp.269-271, 1959.

[6] P. Erdös and A. Rényi. On the Evolution of Random Graphs. Publications of the Mathematical Institute of the Hungarian Academy of Sciences, vol.5, pp.17-61, 1960.

[7] R. Albert and A.-L. Barabási. Topology of Evolving Networks: Local Events and Universality. Physical Review Letters, vol.85, no.24, Dec. 2000.

[8] R. Bellman. On a Routing Problem. Quarterly of Applied Mathematics, vol.16, no.1, pp.87-90, 1958.

[9] J. B. Orlin, K. Madduri, K. Subramani, and M. Williamson. A Faster Algorithm for the Single Source Shortest Path Problem with Few Distinct Positive Lengths. Journal of Discrete Algorithms, vol.8, no.2, pp.189-198, June 2010.

[10] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In Proc. of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX'06), Miami, Florida, USA, pp.129-143, Jan. 2006.

[11] L. Roditty and U. Zwick. On Dynamic Shortest Paths Problems. Algorithmica, March 2010 (published online).

[12] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. Theoretical Computer Science, vol.312, no.1, pp.47-4, Jan. 2004.

[13] Y. Han. A note of an $O(n^3/\log n)$ time algorithm for all pairs shortest paths. Information Processing Letters, vol.105, no.3, pp. 114-116, 2008.

[14] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast Shortest Path Distance Estimation in Large Networks. Proc. of the 18th ACM Conf. on Information and Knowledge Management (CIKM 2009), Hong Kong, China, pp.867-876, Nov. 2009.

[15] D. J. Watts, S. H. Strogatz. Collective dynamics of small-world networks. Nature, vol.393, no.6684, pp.440-442, 1998.

[16] A. L. Barabási, R. Albert. Emergence of scaling in random networks. Science, vol.286, no.5439, pp.509-512, 1999.

[17] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. Proc. of the 39th Annual ACM Symposium on Theory of Computing (STOC'07), San Diego, California, USA, pp.590-598, June 2007.